Understanding the classical model for linking, groundwork: The algorithm

devblogs.microsoft.com/oldnewthing/20130107-00

January 7, 2013

Raymond Chen

The classical model for linking goes like this:

Each OBJ file contains two lists of symbols.

1. Provided symbols: These are symbols the OBJ contains definitions for.
2. Needed symbols: These are symbols the OBJ would like the definitions for.

(The official terms for these are *exported* and *imported*, but I will use *provided* and *needed* to avoid confusion with the concepts of exported and imported functions in DLLs, and because *provided* and *needed* more clearly captures what the two lists are for.)

Naturally, there is other bookkeeping information in there. For example, for provided symbols, not only is the name given, but also additional information on locating the definition. Similarly, for needed symbols, in addition to the name, there is also information about what should be done once its definition has been located.

Collectively, provided and needed symbols are known as *symbols with external linkage*, or just *externals* for short. (Of course, by giving them the name *symbols with external linkage*, you would expect there to be things known as *symbols with internal linkage*, and you'd be right.)

For example, consider this file:

```
// inventory.c


extern int InStock(int id);


int GetNextInStock()
{
  static int Current = 0;
  while (!InStock(++Current)) { }
  return Current;
}
```

This very simple OBJ file has one provided symbol, `GetNextInStock`: That is the object defined in this file that can be used by other files. It also has one needed symbol, `InStock`: That is the object required by this file in order to work, but which the file itself did not provide a definition for. It's hoping that somebody else will define it. There's also a symbol with internal linkage: *Current*, but that's not important to the discussion, so I will ignore it from now on.

OBJ files can hang around on their own, or they can be bundled together into a LIB file.

When you ask the linker to generate a module, you hand it a list of OBJ files and a list of LIB files. The linker's goal is to *resolve* all of the *needed symbols* by matching them up to a *provided symbol*. Eventually, everything needed will be provided, and you have yourself a module.

To do this, the linker keeps track of which symbols in the module are resolved and which are unresolved.

- A resolved symbol is one for which a provided symbol has been located and added to the module. Under the classical model, a symbol can be resolved only once. (Otherwise, the linker wouldn't know which one to use!)
- An unresolved symbol is one that is needed by the module, but for which no provider has yet been identified.

Whenever the linker adds an OBJ file to the module, it goes through the list of provided and needed symbols and updates the list of symbols in the module. The algorithm for updating this list of symbols is obvious if you've been paying attention, because it is a simple matter of preserving the invariants described above.

For each *provided* symbol in an OBJ file added to a module:

- If the symbol is already in the module marked as *resolved*, then raise an error complaining that an object has multiple definitions.

- If the symbol is already in the module marked as *unresolved*, then change its marking to *resolved*.
- Otherwise, the symbol is not already in the module. Add it and mark it as *resolved*.

For each *needed* symbol in an OBJ file added to a module:

- If the symbol is already in the module marked as *resolved*, then leave it marked as *resolved*.
- If the symbol is already in the module marked as *unresolved*, then leave it marked as *unresolved*.
- Otherwise, the symbol is not already in the module. Add it and mark it as *unresolved*.

The algorithm the linker uses to resolve symbols goes like this:

- Initial conditions: Add all the explicitly-provided OBJ files to the module.
- While there is an unresolved symbol:
    - Look through all the LIBs for the first OBJ to provide the symbol.
    - If found: Add that OBJ to the module.
    - If not found: Raise an error complaining of an unresolved external. (If the linker has the information available, it may provide additional details.)

That's all there is to linking and unresolved externals. At least, that's all there is to the classical model.

Next time, we'll start looking at the consequences of the rules for classical linking.

Sidebar: Modern linkers introduce lots of non-classical behavior. For example, the rule

> If the symbol is already in the module marked as *resolved*, then raise an error complaining that an object has multiple definitions.

has been replaced with the rules

> If the symbol is already in the module marked as *resolved*:
> - If both the original symbol and the new symbol are marked `__declspec(selectany)`, then do not raise an error. Pick one arbitrarily and discard the other.
> - Otherwise, raise an error complaining that an object has multiple definitions.

Another example of non-classical behavior is dead code removal. If you pass the `/OPT:REF` linker flag, then after all externals have been resolved, the linker goes through and starts discarding functions and data that are never referenced, taking advantage of another non-classical feature (packed functions) to know where each function begins and ends.

But I’m going to stick with the classical model, because you need to understand classical linking before you can study non-classical behavior. Sort of how in physics, you need to learn your classical mechanics before you study relativity.

Raymond Chen

Follow